TDRK12 Data Structures, 7.5 credits, Spring 2022

Lab: Trees–Task1, 22 February 2022

Due: 8 March 2022

1 Task: Deletion Operation on AVL Trees

1.1 Introduction

In the first task you will create a function to perform deletion of an element in *the right sub-tree* of an AVL tree. A quick recap of deletion in AVL trees (slides 50-53 of the lecture):

- Deletion of a node in an AVL tree is similar to that of binary search trees.
- Deletion may disturb the AVLness of the tree, so to re-balance the AVL tree we need to perform rotations.
- If node is in the right sub-tree, R rotation is performed
- There are R0, R-1 and R1 rotations

To implement the rotations, we will utilize the similarity of rotations after deletion and insertion:

- R0 rotation is similar to LL rotation (see Fig. 1).
- R1 rotation is similar to LL rotation (see Fig. 1).
- R-1 rotation is similar to LR rotation (see Fig. 2).

There are two files with the C source code that we will reuse in this lab:

1. File `ex_bst_9.c` from the exercise contains two functions that are needed for deletion
   a) `struct node *deleteElement(struct node *tree, int val)`
   b) `struct node *findLargestElement(struct node *tree)`
2. File `avl_tree_insert.c` from the textbook (additional comments explaining how insertion works) contains functions needed for insertion:
   a) `struct node *insert(int data, struct node *tree, bool *ht_inc)`
   b) `struct node *search(struct node *ptr, int data)`

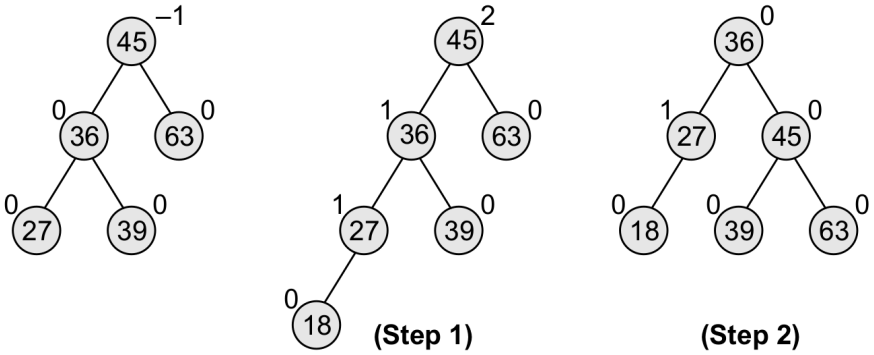

Figure 1: LL rotation after insertion of 18

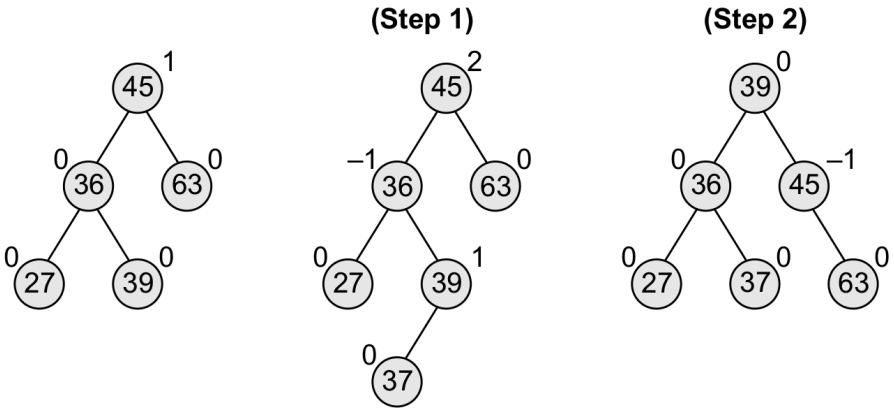

Figure 2: LR rotation after insertion of 37

c) `void display(struct node *ptr, int level)`

Note: this task is not heavy on the coding part but it will require effort to achieve conceptual understanding of how insertion and deletion operations work with AVL trees.The best way to do this is to study both code and illustrations (in the slides or textbook) together and, at the same way, do simple drawings on paper.

1.2 Implementation Directions

To carry out this task, the following steps are recommended:

1. Study the supplied code in `avl_tree_insert.c`.
   - Read the code. Detailed comments are added there with references to the lecture slides.
   - Observe how the shape of a resulting AVL tree depends on the order of entering node values. Run the code several times with different inputs and the same inputs but in different order.

- Examine how the AVL tree is re-balanced after insertion. Use examples in Fig. 1 and Fig. 2.

2. Implement deletion of a node in the *right sub-tree* of an AVL tree. In this task we will work only with the right sub-tree.
   - You will need to add to the code in the `avl_tree.c` file these functions
     - `struct node *delete(int data, struct node *tree, bool *ht_inc)`
     - `struct node *findLargestElement(struct node *tree)`
   - The general structure of the `delete()` function should be similar to the one from `ex_bst_9.c`
   - To re-balance the AVL tree after the deletion, perform a rotation: R0, R-1 or R1 rotation.
   - Employ the similarities between insertion rotations and deletion rotations. Reuse the code from `avl_tree_insert.c` for LL and LR rotations to implement R1, R0, and R-1 rotations. Refer to slides 50-53 if necessary.
   - Take into account that **R rotations** are performed after deletion in **the right sub-tree**, however, you need to reuse *LL and LR rotations* that are performed after insertion in *the left sub-tree*. So be careful when you copy and modify the code.
   - One more hint: The value of the `ht_inc` variable that controls re-balancing (see the comments for explanation) needs NOT to be changed in the branch `case -1: /* Right heavy */` after deletion because -1 will be on path to critical node and re-balancing will be needed at a higher level.

3. Modify the main() function, to test the deletion operation with two test cases from the textbook. Fig. 3 for R-1 rotation (10.53 from the textbook) and Fig. 4 for R0 rotation (modified 10.54 from the textbook).
   - Create two `int` arrays of fixed size with the required nodes. Make sure that the node values are in *the right order* to create the trees from the test cases.
   - Do insertion of the nodes in a loop.
   - Then Display a menu to delete a node and display the tree after the deletion.

4. Test and debug the code using the two test cases until your solution provides an output similar to the one given below

1.3 Test Cases for Deletion in AVL Tress

There are two test cases that you need to use to check your code. The input AVL trees are shown in Fig. 3 and Fig. 4. The output is detailed in Sec. 1.4.

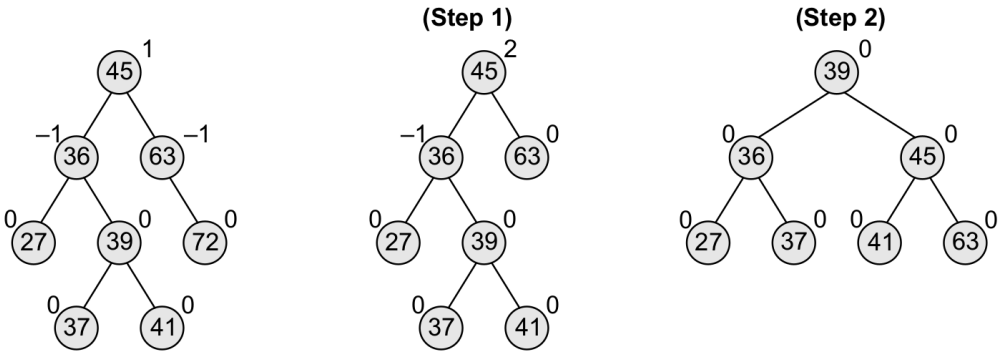

Figure 3: Example of R-1 rotation after deletion of 72 (fig. 10.53)

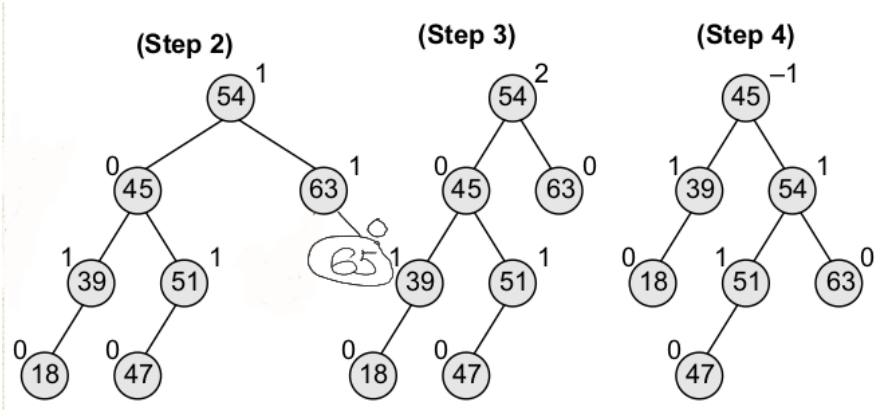

Figure 4: Example of R0 rotation after deletion of 65 (modified fig. 10.54)

1.4 Deliverables

The deliverable is working source code that includes the required function
`struct node *delete(int data, struct node *tree, bool *ht_inc)`.
It is up to you to decide how you code the `main()` function but the output should be similar to the one below:

An example of execution of the first test case:

Output for the first test case

```
Running first test
The fisrt AVL tree is:

   72
  63
 45
    41
   39
    37
  36
   27

1.Delete
2.Display
3.Second test
Enter your option : 1
Enter the value to be deleted : 72
R-1 rotation
1.Delete
2.Display
3.Second test
Enter your option : 2
Tree is :

   63
  45
   41
 39
   37
  36
   27
```

An example of execution of the second test case:

Output for the second test case

```
Running second test
The second AVL tree:

   65
  63
 54
   51
    47
  45
   39
    18

1.Delete
2.Display
3.Quit
Enter your option : 1
Enter the value to be deleted : 65
R0 Rotation
1.Delete
2.Display
3.Quit
Enter your option : 2
Tree is :

   63
  54
   51
    47
 45
  39
   18
```